A Situated Evaluation of the Object Management Group's (OMG) Object Management Architecture (OMA)

Evan Wallace
National Institute of Standards and Technology
Manufacturing Engineering Laboratory
Gaithersburg, MD, 20899, USA
wallace@cme.nist.gov
(301) 975 3520
(301) 258 9749 FAX

Kurt C. Wallnau
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213, USA
kcw@sei.cmu.edu
(412) 268 3265
(412) 268 5758 FAX

Abstract

It has been difficult to objectively assess the real value or maturity of the Object Management Group's Object Management Architecture (OMA). While experience reports have appeared in the literature, these have focused more on the functionality of the end-system than on systematically exploring the strengths and weaknesses of the OMA, and providing practical guidelines on the effective use of the OMA for specific software-engineering problems. In this paper we describe a case study in the use of the OMA to integrate legacy software components into a distributed object system. We assess the OMA in this problem context, and indicate strengths and weaknesses of the specification and current implementations. We extrapolate our experience to a broader class of component-based software systems, and recommend an architectural strategy for the effective use of the OMA to this class of systems.

Category: experience, software engineering practice.

1. Introduction

The Object Management Architecture (OMA) [1] continues to attract attention, with numerous implementations of the OMA common object request broker architecture (CORBA)[1] emerging in the commercial marketplace. How does an organization decide whether to embrace this technology? The stakes are high since the OMA can have a profound influence on the design and implementation of application software. A technology assessment strategy that can identify the value-added of a new technology, and simultaneously reveal how best to exploit this value added, is therefore of great potential value.

We have applied one such evaluation technique to the OMA [2]. We describe the evaluation technique as "situated" because it describes the technology being evaluated in terms of peer technologies (in order to identify the new "value-added" features of a technology, i.e., its *feature delta*), and then places the feature delta into specific usage contexts. The technology is thus situated in both the technology marketplace and in the problem domain in which the technology is to be evaluated.

There are many ways of evaluating feature deltas. In some cases it is possible to isolate and benchmark elements of the feature delta, as illustrated by the comparative benchmarking of CORBA and RPC [3]. However, the lack of a specific problem context within which to evaluate the benchmarks can limit the effectiveness of this form of evaluation. To overcome this limitation, experimentally-focused case studies can be undertaken that apply the feature delta to representative problems of an application domain. Such case studies can be particularly fruitful as in addition to providing a problem context for evaluating a technology, then can also provide a wealth of practical experience in how to best apply a feature delta to these problems.

In this paper we describe one experimentally-motivated case study in the use of the OMA to an increasingly-important problem domain: the integration of component-based systems (systems comprised of stand-alone, independently-executing software packages). Section 2 provides background information on the engineering problems inherent to component-based systems. Section 3 provides further background on the manufacturing domain, and on the legacy collection of computer-aided design engineering (CADE) components that we modernized into a distributed, object-based system. Section 4 describes the impact of the OMA on the architecture of the modernized system, and Section 5 continues the discussion of the impact of OMA at a more detailed implementation level. Finally, Section 6 states our conclusions about the OMA based on this experiment.

2. Background: Component-Based Systems

While all (real) systems are composed of components, in our usage *component-based* systems are comprised of multiple software components that:

1. We use CORBA to refer to only the message broker component of the OMA; we use OMA to refer to CORBA plus additional OMA services.

- are ready "off-the-shelf," whether from a commercial source (COTS) or re-used from another system;
- are self-contained and possibly execute independently;
- will be used "as is" rather than modified;
- must be integrated with other components to achieve required system functionality; and,
- have significant aggregate functionality and complexity.

Examples of component-based systems can be drawn from many domains, including: computer-aided software engineering (CASE), design engineering (CADE) and manufacturing engineering (CAME); office automation; workflow management; command and control, and many others.

2.1 Architectural Mismatch: The Core Issue

In contrast to the development of other kinds of systems where system integration is often the tail-end of an implementation effort, in component-based systems determining how to integrate components is often the only latitude designers have. In our evaluation we were interested in whether the OMA suggested solutions to the core engineering problems in the integration of component-based systems: *architectural mismatch* [4].

The term "architectural mismatch" refers to the problem that components always embed assumptions about their intended operational context, and these assumptions often conflict with assumptions made by other components. For example, a component that uses a graphical human interface as the sole means of executing component functions has embedded an assumption that will render it unusable in a system that must run in batch mode—unless this mismatch can be removed by some form of component adaptation. Scores of other kinds of mismatches are commonplace involving, e.g., multi-users support, resource management, and security.

The term "architectural mismatch" implies more than just mismatched component assumptions: it also implies that mismatches can arise between components and a *software architecture* [5]. The general consensus is that software architecture deals with high-level design patterns[1] for structuring and expressing system designs. A sense of what is meant by "high level" is that these patterns are often expressed in terms of components, connectors and coordination. *Components* refer to units of functionality[2], *connectors* refer to the integration of components, and *coordination* is the manner in which components interact at run-time. Architectural mismatches can arise that inhibit component integration and coordination.

Not all architectural styles are equally-well suited to a specific design problem. Many factors can influence the selection of a style—functional requirements, quality attributes (e.g., modifiability) and *a priori* design commitments (e.g., distributed system). Thus, a given set of software components may be assembled into a number of architectural styles, and may exhibit different kinds of architectural mismatches in each architectural setting. This suggests a reference model for describing the engineering practices involved in assembling component-based systems, as depicted in Figure 1.

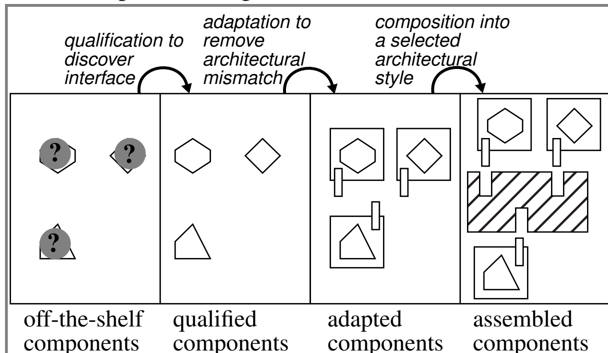

Figure 1. Architectural Assembly of Components

The vertical partitions depicted in Figure 1 describe the central artifact of component-based systems—the components—in various states:

- Off-the-shelf components have hidden interfaces (using a definition of interface that encompasses all potential interactions among components, not just an application programming interface[6]).
- Qualified components have discovered interfaces so that possible sources of architecture mismatch have been identified. This is (by the definition of interface) a partial discovery: only those interfaces that mismatch an architectural style or other components are identified.
- Adapted components have had their architectural mismatches ameliorated. The figure implies a kind of component "wrapping," but other approaches are possible (e.g., the use of mediator agents).
- Assembled components have been integrated into an architectural infrastructure. This infrastructure will support component assembly and coordination, and differentiates architectural assembly from ad hoc "glue."

1. The terms "styles" and "idioms" are synonymous with patterns.
2. The component-based definition of this term, which is the one used here, is more a restrictive form than that used in software architecture literature.

2.2 A Component-Based System Evaluation Context for the OMA

Although the reference model depicted in Figure 1 is simplistic, it is nonetheless sufficient to suggest the following questions for an evaluation of the OMA:

- Are certain architectural styles suggested by the OMA? If yes, does the OMA provide an adequate mechanism for implementing these styles? Are other mechanisms in addition to the OMA (as currently specified) required?
- Does the OMA introduce potential sources of architectural mismatch beyond those implied by architectural style? If yes, do these result from the OMA specification, or peculiarities of commercial implementations of the OMA?
- Are some kinds of components more readily adaptable to the OMA than others? If yes, what are the characteristics of components that make them more adaptable? What adaptation mechanisms work best with the OMA?

The case study described in detail, below, provided significant insight in answering these questions.

3. Background on CADE

While the nature of component-based systems provides a needed backdrop for focusing the OMA technology evaluation, the problem setting must be completed with requirements stemming from an application domain, and those stemming from the particular problem being addressed. We selected the manufacturing domain as a basis for this case study.

3.1 A Manufacturing Domain of the Future

The manufacturing processes required to move a product from concept to realization are many and varied, and often require the application of highly-specialized skills and computing resources. Job scheduling, shop layout, and manufacturability analysis are examples of such skills that are supported by software technology. In many cases these specialized skills are relatively independent of the underlying application domain—manufacturability analysis techniques for automotive parts and washing machine parts are quite similar. Some believe that a breakthrough in manufacturing efficiency can be achieved if these “horizontal” skills can be freed from their existing “vertical” market confinements, and allowed to develop in the free-market.

The challenge is how to re-assemble these horizontal specialties into *virtual enterprises*, i.e., otherwise independent manufacturing companies collaborating in vertical manufacturing enterprises. Virtual enterprises are a means of supporting the flexible combination of the skills and tools from many highly-specialized companies; from this, faster market response time, reduced time-to-market, and increased manufacturing quality can be achieved. However, in addition to regulatory and business-model challenges of virtual enterprises, additional technology infrastructure is needed that will support:

- the integration of separately-developed, specialized, computer-aided manufacturing technologies;
- geographical distribution of computing resources, and support for heterogeneous computing environments;
- fee-for-service brokering of computer-based services to enable competition for specialized tools and skills.

In short, virtual enterprises in an increasingly specialized manufacturing world will rely more and more upon information technology such as supported by distributed object technology. However, the existing investment in computer-aided technology will need to be preserved, and adapted, to exploit distributed object technology.

3.2 A Legacy Manufacturing System

We needed a legacy manufacturing system for modernization to distributed object technology that would be simple enough to quickly prototype, yet sophisticated enough to constitute a reasonable test of the OMA. As a domain model of manufacturing activities suggests [7], there are many subdomains that might provide for fertile hunting. From this model we determined that the *design engineering* activity was suitably focused and automated.

Design engineering involves modeling and simulating part designs to test the performance of parts under certain expected real-world use conditions. This analysis can be used to determine the adequacy of a part design for performing a function, to optimize a part design, and to lower the cost and/or weight of a part while preserving confidence in its performance. A diverse range of software components have been developed that support design engineering, providing the basis for a OMA case study.

The Sandia National Laboratory Engineering Analysis Code Access System (SEACAS) provided us with a representative set of design engineering components. SEACAS supports functions such as problem definition, simulation, analysis, and visualization (see Figure 2). Sandia has developed many components in each functional category, reflecting both the diversity of analysis problems being addressed and the evolving sophistication of design engineering methods. For our case study, we selected a core set of SEACAS components that could represent a single thread through a design engineering scenario.

The general usage scenario for these tools is as shown in Figure 3, which uses a process description formalism similar to IDEF-0: the bubbles represent activities; data input appears on the left of activities, and data output appears on the right; mechanisms appear on the bottom of activities;

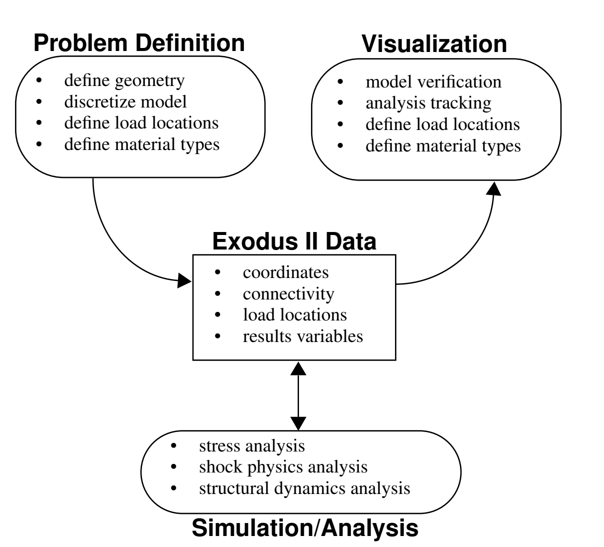

Figure 2. SEACAS Tool and Data Relationships

and, control flow appears on the top of activities. The control flow "end-user" which would appear on each activity is omitted from Figure 3 for simplicity.

Following Figure 3, an analyst uses FASTQ to produce a *discretized*[1] model of the part to be analyzed. The visualizing tool BLOT then can be used to produce a graphical representation of the model for inspection by the analyst. Based on that inspection the analyst either re-runs FASTQ to correct the model or feeds the model to GEN3D to extrude the two-dimensional model into the third dimension. Another inspection/correction iteration may occur for the resulting three-dimensional model before the model is fed to the analysis tool. One or more analyses may be conducted, with results evaluated via the visualization tool or through inspection of other JAS3D output (not illustrated). As suggested by Figure 3, the process can be highly iterative, reflecting a process of convergence on a suitable design.

As Figure 2 points out, many of these tools make use of a common data format (Exodus-II). This is an important property that greatly simplified the prototyping effort. However, there is more to component integration than data interchange. For example, in the scenario described above end-users are responsible for launching tools (often with personalized scripts), managing tool output (in private directories), and preparing the output of one tool for use as input to another (sometimes with manual application of filters since not all SEACAS tools use the same version of the Exodus data format). In effect, the end-user is the system integrator, continually exposed to all of the low-level details and technology

1. A two- or three-dimensional mesh that models the contours of a solid.

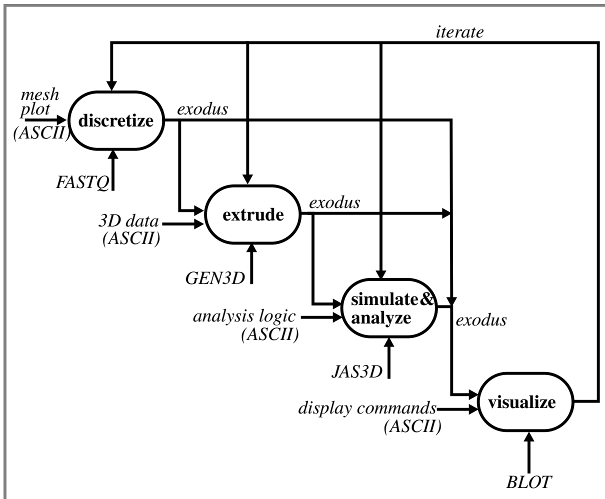

Figure 3. Components and End-User Scenario

dependencies exhibited by the SEACAS components, and the environments in which they operate.

3.3 Objectives for OMA-Based Modernization

From the particular requirements of SEACAS, we determined that the case study should evaluate the use of the OMA to integrate a legacy collection of design engineering tools in order to:

- present a uniform virtual environment for end-users that would reflect the nature of the analysis activity and not the idiosyncrasies of specific SEACAS components;
- support wide-area distribution of SEACAS services while maintaining control over the software that provides these services (some of which contain classified algorithms);
- deliver reasonable, usable and predictable performance to end-users who are otherwise accustomed to using computer services in local area network settings.

These requirements extend and refine those of the underlying manufacturing application domain (Section 3.1).

4. A Distributed Object Architecture for the SEACAS Components

The modernization objectives (cited above) can be thought of as *quality attributes*—externally-visible system properties that deal with issues other than functionality. The significance of quality attributes is that it has been demonstrated that the top-level design, or architecture, of a system is the key factor leading to the satisfaction (or lack) of these non-functional requirements [8]. Thus, for our evaluation of OMA we needed to determine how well it addressed architecture-level issues inherent to component-based systems (Section 2.2) as well as how it addresses component-based systems that exhibit these quality attributes.

In the following discussion it is useful to bear in mind that the purpose of the case study was an evaluation of the OMA feature delta. If the design problem is viewed strictly in terms of the four components integrated, then there were undoubtedly simpler design solutions to achieving the limited objectives outlined in Section 3.3. However, viewing the design problem as a representative one in a broader class of problems led us to design solutions that required a more elaborate use of the OMA, i.e., use of the OMA feature delta.

4.1 Architectural Overview

We quickly discovered that the OMA suggested an architectural approach that makes extensive use of the OMA object model and the CORBA interface definition features; we were pleased to discover that this feature delta beyond the more primitive form of an RPC interface definition had such significance. This central role for an object model in a component-based architecture is illustrated in Figure 4, which depicts a top-level view of the prototype architecture.

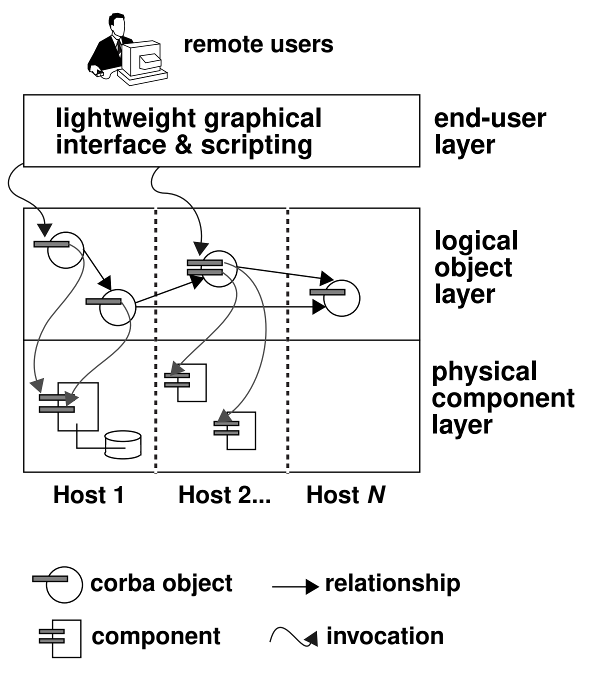

Figure 4. Architectural Overview

This architecture can be described in terms of how it supports integration—the core design activity in component-based systems (architectural mismatch is the inhibitor of integration). One useful way to think of integration is as a *relationship* between two integrated entities, where the relationship has four *aspects*: control, data, process and presentation [9][10]:

- Control integration describes how components make requests of (or invoke) each other's services.
- Data integration describes how components make data available to each other.
- Process integration describes what end-user process is supported by, or activates, the integration relationship.
- Presentation integration describes how end-users interact with the endpoints of an integration relationship.

The end-user layer addresses presentation and process integration through interactive, graphical client interface, and scripting logic that sequences and controls the execution of remote services. This layer will not be described in detail; we view it as an application layer supported by the architecture. However, we note that abstraction mismatches between the scripting language and the object model were introduced by our use of TCL/Tk; a mechanism such as Java, which is object oriented, would address this problem.

The physical component layer addresses the run-time aspects of the SEACAS components, and various platform dependencies. At this level a number of sometimes subtle interactions between components, operating system and CORBA implementation arose. These detailed implementation issues are discussed in Section 5.

The logical object layer addresses data and control integration. Interestingly, we discovered that the OMA suggested an architectural style to address these aspects that in effect blends two different styles: a *repository* style for data integration and a *structural* style for control integration. The remainder of this section will describe these styles and how they were realized with the OMA.

4.2 Object Layer as Data Repository

The repository-style architecture is characterized by a central data repository that is used as a principle means for components to coordinate their execution and share results. Numerous examples of repository-style architectures have been seen in the computer-aided software engineering (CASE) domain [11]; examples in the manufacturing domain are also emerging [12].

The repository-style architecture is motivated by two factors, both of which are relevant to SEACAS:

1. The data artifacts that are manipulated by software components are key assets that must be managed. Mechanisms for access control, versioning, backup and recovery, transactions, etc., are all important for the effective management of data assets.
2. The structure of data can be quite complex, with different kinds of data related in a complex network of aggregation and dependency relationships. Mechanisms

for schema definition and evolution, query, and navigation are all needed to manage this complexity.

The two OMA services that we found to be most important for the SEACAS repository were *relationships* and *persistence*. Relationships were used to define links between various kinds of SEACAS artifacts while persistence allowed networks of object instances to persist beyond user sessions.

We used the object-oriented features of the CORBA interface definition language (IDL) to model SEACAS artifacts in a class hierarchy, and used relationships to express the derivation history from one class of artifact to another. A simplified object model is depicted in Figure 5, which shows the

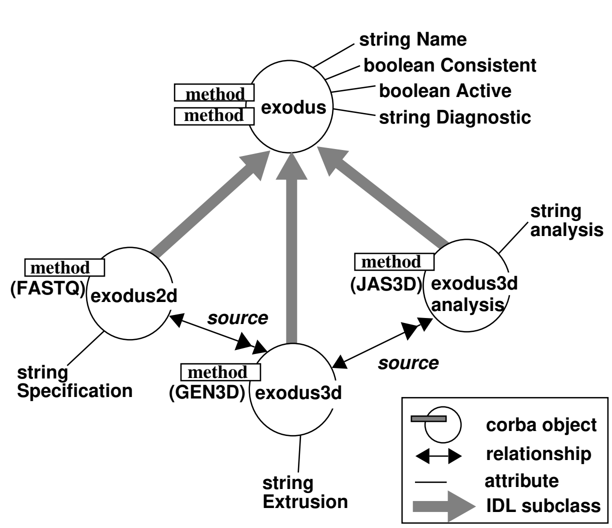

Figure 5. Object Types and Relationships

major object types and their relationships to each other and to the SEACAS components. For example, *exodus2d* (a two-dimensional mesh) inherits (and defines its own) operations and attributes from the *exodus* abstract superclass; *exodus2d* functionality is implemented by *FASTQ*; and, it participates in a 1:many relationship with *exodus3d* objects to indicate that a single two-dimensional mesh can be extruded into several three-dimensional meshes.

The net effect of populating the logical object model layer with instances of these object types and relationships is to create a distributed object repository of SEACAS objects. On the surface, this would seem to indicate that the OMA provides a good foundation for repository-style architectures, and in the large this is true. Upon looking deeper, however, there are limitations to the OMA specification and commercial implementations of the OMA that may effect the scaleability and robustness of OMA-based data repositories.

With respect to commercial implementations, vendors are not required to implement any of the OMA services beyond CORBA. The object request broker we used did not support the relationship service, and it supported a non-standard persistence service. These missing services resulted in a substantial increase in programming complexity and a decrease in application functionality and robustness. For example, we implemented relationships as object-valued attributes (object references); this required additional encoding of dependency management logic within *exodus* methods, and introduced subtle interactions with the persistence mechanism, which itself was quite complex. Also, the bi-directionality, arity and object type constraint checking, and compound object operations that would have been available with OMA relationship services were too expensive to implement.

Issues are also raised by the specification of the OMA services. For example, the underlying OMA assumption that object services can be separately specified and individually implemented is a cause for concern. There are, for example, specification and implementation dependencies between persistence and transactions, naming and security, and relationships and life cycle. While the OMA specification is sensitive to some of these dependencies, many more dependencies exist than can be conveniently specified or even anticipated. The PCTE specification—a quasi-object based repository technology, is a broad indication of the level of complexity involved [13]. Other specification issues that will inhibit the development of robust OMA-based repositories include: the lack of various data management services (e.g., schema definition, administration, object ownership and sharing), and optimistic design assumptions about object granularity and network/inter-process communication performance.

However, while the OMA does have limitations regarding data management services, component-based systems might not, in general, require repositories that implement a full range of database management functionality. For our case study even simple OMA services—had they been available—would have sufficed. Thus, we concluded that

- Object services are both useful and essential for component-based systems integration; and,
- the OMA provides (barely) sufficient data management services, provided the designer is not over-exuberant.

4.3 Object Layer as Structural Architecture

The repository style architecture addresses issues of data and object management, but it does not address how the functionality of components (such as the SEACAS components) is mapped to persistent objects, nor how these objects interact at run-time (the coordination model). There are two overall approaches to addressing these issues—a *functional* approach and a *structural* approach.

The functional approach is by far the predominant approach to component-based systems. This approach defines component interfaces in terms of their specific functionality. Functional architectures are good for describing system functionality and for integrating specific functionality but are weak at addressing the run-time properties of a design. e.g., throughput, latency and reliability.

The structural approach has emerged as the study of software architecture has intensified. Rather than defining component interfaces in terms of functionality, structural styles define interfaces in terms of the role a component plays in a *coordination model*—a model that describes how the components interact. A simple illustration of a structural style is UNIX pipes and filters; more sophisticated illustrations include structural models for flight simulators [14], and the Simplex architecture for evolvable dependable real-time systems [15]. The structural approach has become more popular recently because it yields architectures that, by definition, support analysis of dynamic system properties.

CORBA IDL supports functional and structural styles equally well. However, we discovered that the structural approach is particularly well-suited to component-based systems; further, it is also suggested by the OMA. This conclusion can be demonstrated by a discussion of the simplified structural architecture depicted in Figure 6, which illustrates the key coordination interfaces of one type of SEACAS object[1]. The focus of attention for this discussion is on the structural model in Figure 6.

The essence of the SEACAS coordination model is that objects are data managers that are either in a consistent or an inconsistent state. A consistent object is one whose *input attribute* is consistent with the EXODUS data that has been derived from this attribute through the execution of a SEACAS component. The input attribute for the SEACAS object depicted in Figure 6 is *analysis*, which is a string containing simulation instructions for the finite element analyzer. Clients of an object can test whether an object is consistent using the *consistent?* probe, and can re-establish consistency by using the *update* method. The *update* method is non-blocking; clients can determine if an update is still in progress by using the *done?* probe. Clients can examine the results of an update in two ways: through a *view* method on the SEACAS object (not illustrated), or through an event queue. The *view* method can only be used on a consistent object, while the event queue can be used while an update is in progress (an object is not consistent until the update has successfully completed).

Our conclusion that structural styles are particularly well-suited for component-based systems and for the OMA is based on these two observations:

1. The interfaces of other SEACAS objects are nearly identical.

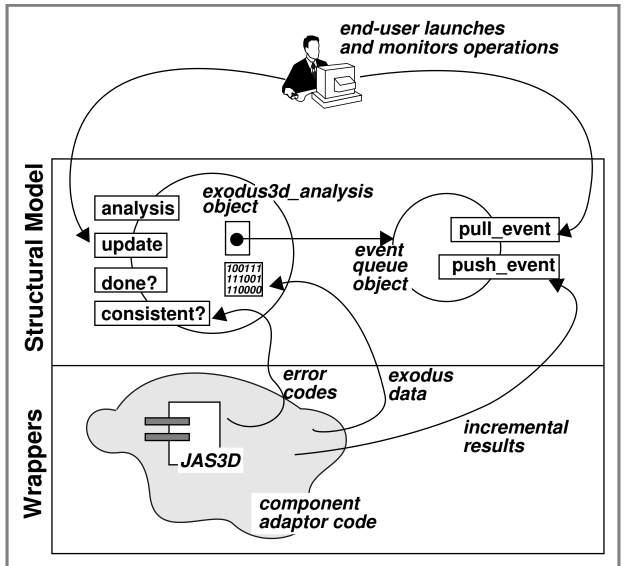

Figure 6. Structural Architecture (Overview)

1. Explicitly representing the coordination model via object interfaces addresses ambiguous and restrictive run-time semantics inherent in the OMA specification as described below.
2. Coordination-focused object interfaces help identify areas of architectural mismatch, and suggest re-usable adaptation techniques for different kinds of mismatch.

The first is discussed here, the second in Section 5.

The decision to make *update* a non-blocking method represents a departure from more straightforward use of CORBA features for client-side concurrency. Typically, pure[2] clients will use CORBA's dynamic invocation interface (DII)[3] if they do not wish to block on a remote method; alternatively, multi-threaded clients could create a separate thread for each blocking method. Implementing a non-blocking *update* method appears to restrict the client's options—so why do it? There are two reasons.

The first reason is that finite element analysis may consume anywhere from a few seconds to several days of wall clock time. Relying on a synchronous connection over a wide-area network for such durations will do violence to our reliability requirements—a momentary network failure would cause an update to fail. Making the *update* method a *oneway* call—another CORBA mechanism—is inadequate because this mechanism does not permit clients to be notified of the many exceptional circumstances that might indi-

2. Pure clients are applications using but not containing any objects; object implementations can also be clients, but are not "purely" clients.
3. The DII provides mechanisms for deferred synchronous communication, aka asynchronous polling.

cate a problem with the update prior to the actual execution of SEACAS component services. Thus, had the update method been implemented using default CORBA synchronous logic, clients would have been forced to use the DII to achieve the desired level of system reliability. This is an unfair burden to place on clients because of the additional overhead forced on the client to dynamically build all operation requests; also, the architecture should ensure the reliability regardless of the form of client interface used.

The second reason concerns the way the CORBA basic object adaptor (BOA) addresses server-side concurrency, i.e., how to achieve concurrent execution of the methods of one or more objects within a server process. Clearly, servers that can exploit thread libraries will have a ready-made mechanism; however, the Object Management Group is loath to build implementation dependencies such as this into their specifications. Thus, the BOA specifies *activation policies* that (in increasing concurrency) associate a server process with:

- classes of objects ("shared" activation policy);
- individual objects ("un-shared" activation policy); or,
- individual methods ("per-method" activation policy).

By implementing *update* as non-blocking we subverted the BOA activation policy, since non-blocking semantics requires *de facto* concurrency of object implementations.

However, the un-shared and per-method policies require interprocess communication (IPC) for objects to invoke each other's methods. Certainly this kind of coupling is to be expected where different object types are closely in a class hierarchy, as is the case with SEACAS objects (refer to Figure 5). While it is possible to specify multiple sets of interfaces for objects—those for clients and "private" interfaces for friends, this can require substantial additional coding and in any event does not address the cost of the IPC or the creation of separate processes, especially in the case of per-method activation. Also, implementations of CORBA treat activation policy as a kind of configuration option for object implementations: different installations of the services may choose different policies. Again, such important system properties should be reflected in the architecture of the system, not in implicit coordination semantics and configuration options.

A final point on this topic is that these design and implementation decisions could have been taken even in a functional style, i.e., had the update method been a direct interface to a specific SEACAS function. However, the structural approach makes these coordination model decisions explicit in the object interface—there is no mistaking the assumptions concerning concurrency in the objects described in Figure 6. Moreover, the structural approach addresses the coordination model the same way for each SEACAS component, as is evident in their common interfaces; this would be far less obvious in a functional style.

5. Component Adaptation Issues

As illustrated in Figure 6, there are several distinct relationships between the SEACAS components and the structural architecture. Each of the connections between component and structural model indicate an interaction between the architecture and a component, and hence an area of potential mismatch.

For example, from Figure 6, the *consistent?* probe should return "true" if and only if the SEACAS component executed properly. A failure could result from either a semantic fault or a system fault. SEACAS assumes that end-users will determine the success of an operation by reading diagnostic output; this does not match with the structural model, which assumes a more automated approach. System faults, such as a component crash, can also arise. SEACAS assumes such crashes will be evident since the end-user will have directly invoked the component; this also does not match with the structural model, which hides the component and its invocation.

Removal of these and other kinds of mismatches requires some form of component adaptation. In the above example, the diagnostic output needed to be parsed to determine if, and what kind, of semantic fault arose (if any, since the same output stream was used to report success and failure); also, the component process needed to be monitored for exceptional conditions and exit codes. Each of the other component-to-architecture connections depicted in Figure 6 exhibited such mismatches that needed to be resolved by component adaptation code (the "wrappers" in Figure 6)—there were other areas of mismatch that were not depicted in the figure for reasons of clarity.

The term "wrapper" is very misleading—the term implies that component adaptation is accomplished through action taken on the component itself, i.e., encapsulating the component behind a veneer that presents an alternative, translated interface. However, this is just one approach to removing architectural mismatch. A better way of thinking of component adaptation is to observe that the mismatch occurs between two entities, in this case a component and an architecture, and that adaptation can occur at either or both ends of the relationship, or in the middle via an intermediary agent.

We do not have a complete model of adaptation techniques. However, the structural model did suggest a categorization of types of architectural mismatch that could arise, and our implementation provided at least one technique for addressing these mismatches. This confirms our earlier assertion that structural styles help focus attention on key areas of architectural mismatch. Also, since the structural model is quite general—there is little about Figure 6 that implies dependencies on the manufacturing domain—there is reason to hope for the development of architecture-specific adaptation techniques.

Unfortunately, while well motivated, this last hope may be thwarted by the complexity of the adaptation, especially where the adaptation involves the architecture-side. We discovered that architecture-side adaptation is characterized by a thorny tangle of interactions between the coordination model, OMA semantics, vendor-specific features of the object request broker (ORB), operating system primitives, and characteristics of the components themselves. While a complete exposition of these issues would require ‘a code walk-through, a high-level overview of one example may reveal the nature of this complexity. We take the accumulation of incremental output on an event queue, as illustrated in Figure 6, as our example.

Recall that the decision to make the update method non-blocking in effect mandated that the object server support concurrent execution of object services, and that we could not use the BOA per-object or per-method activation policy for reasons discussed earlier. To this we add that the ORB implementation we used for the case study did not support multi-threaded servers. As a consequence, we were forced to implement our own “homegrown” concurrency service. Those familiar with UNIX systems programming will not be surprised by our approach to this problem, and Figure 7 depicts the key elements of our solution.

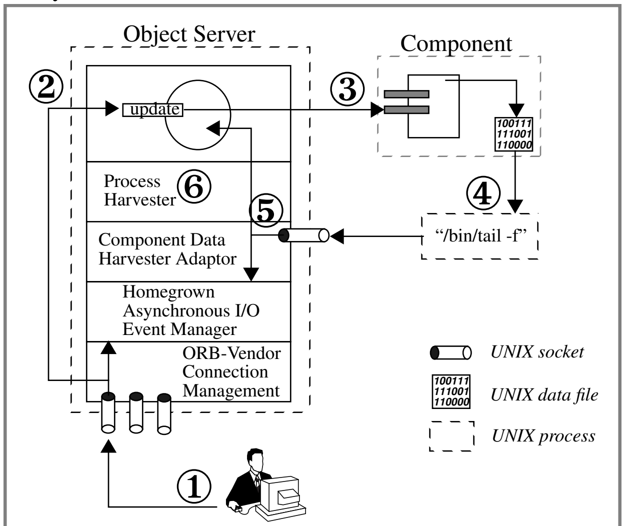

Figure 7. Component Adaptation (Detail)

The core of the solution makes use of UNIX asynchronous I/O and sockets. The idea is to have the main event loop in the object server “listen” (via the *select()* system call) for activity on any number of socket connections, and when activity is detected on a socket invoke a callback procedure that is appropriate to the kind of activity detected (e.g., data available). With this background in mind, the solution works as enumerated in the figure:

1. A client inititiates connections to objects via sockets. These sockets are installed as they are created into the asynchronous event handler so that client requests for object services can be detected and dispatched. This installation depends upon ORB-vendor connection management services that allow the detection of new client connections, and the use of sockets.
2. A client request for an *update* is detected as activity on the client socket. The callback routine registered to handle this activity calls the ORB-vendor’s implementation of the standard BOA object event handling method; this is portable across ORBs, provided the ORB allows server developers to access these routines (which is not true of one ORB we have used).
3. In response to the *update* request the SEACAS component is launched (via *fork()* and *exec()*). Different kinds of components may require different approaches to this step, for example the component could be a server that must be connected-to rather than launched. The process identifier of the launched tool is installed in the process harvester, and signal handlers are established to monitor the state changes of the process.
4. The component will begin writing its incremental output to a data file; component-side adaptation ensured that the file name was unique to each invocation. A monitor process is launched to detect state changes to the output file and report them to a socket established in step 3 for this purpose; this requires delicate timing logic because the data file will appear an indeterminate time after the SEACAS component is launched.
5. Incremental output is detected on the data harvester socket. A callback procedure is invoked by the event manager to parse the data, since only portions of the data file that are being generated by the SEACAS component are of interest for the purposes of observing the progress of the finite element analysis. The parsed data is enqueued on an event channel.
6. The process harvester detects the termination of the SEACAS component, and determines whether the termination was normal or exceptional (needed to determine if the object is in a consistent state). Upon termination of the SEACAS component, the data monitoring process is terminated, sockets are closed, and I/O callbacks removed from the event handler.

This illustration, while gory in detail, serves to highlight a number of important points. First, adapting architectural mismatch may require low-level, intricate code. Techniques for making this process rational and repeatable will contribute greatly both to programmer productivity and system reliability: most of the problems our prototype experienced involved low-level adaptation code. Second, vendor-specific ORB features have an overwhelming influence on adaptation techniques. Another commercial ORB we have used requires completely different, but not less complex, adaptation approaches. Last, the possibility of developing architecture-specific adaptation techniques may be hampered by intricate ORB, tool, and operating system dependencies. However, this is only an issue if portable object implementations is desired—and the OMA does not support object implementation portability in any event.

6. Conclusions about the OMA

In Section 2.2 we posed a specific range of questions that the OMA evaluation would answer. First, the OMA does indeed suggest a particular architectural style, which we referred to as a repository style in this paper. However, we are skeptical that the key OMA services (e.g., persistence, relationship, transactions and relocation services, to name just a few) will be sufficiently well-integrated and functional to implement robust distributed database management functionality. Despite this skepticism, we found the OMA to be sufficiently flexible and expressive to describe a wide range of other styles, including hybrid styles that make selective use of OMA distributed object management services.

Second, we found that the OMA does introduce its own forms of architectural mismatch. This is to be expected—most legacy components will not have been designed to operate within the context of a distributed object model. On the other hand, we were surprised at how sensitive our component adaptation tactics were to specific ORB features. In our prototype, vendor-specific features played a key role, in part because the vendor did not provide standard implementations of needed OMA services (e.g., persistence). Vendor-specific features also played a role in low-level code that dealt with mapping between the operating system process model and the OMA object model.

Last, although we were not able to identify component characteristics that are useful for OMA-based integration beyond the usual characteristics of integrable components (open interfaces, etc.), we can state categorically that developers should be prepared to write low-level, and often intricate code to enable the ORB to launch and manage the execution of legacy components. Not only must the component's implicit coordination model be reconciled with the OMA object model and the application architecture, but the mapping of the operating system process model (and other platform-specific resource models) to the OMA object model must also be addressed.

Despite the somewhat negative tone of these conclusions, we are overall quite impressed with the applicability of the OMA to distributed component-based systems. However, while the OMA does make the building of distributed component-based systems easier, it does not make the hard design and implementation decisions involved in such systems disappear. Nevertheless, we believe the OMA provides sufficient mechanisms and latitude for system designers to address many of these difficult challenges.

7. References

[1] *Object Management Architecture Guide, Revision 2.0*, Second Edition, OMG TC Document 92.11.1, Object Management Group, 492 Old Connecticut Path, Framingham, MA, 01701.

[2] Brown, A., Wallnau, K., "A framework for systematic evaluation of software technologies" IEEE Software, September 1996.

[3] Wallnau, K., Rice, J., "ORBS In the Midst: Studying a New Species of Integration Mechanism", in Proceedings of International Conference on Computer-Aided Software Engineering (CASE-95), Toronto, CA, July 1995.

[4] Garlan, D., Allen, R., Ockerbloom, J., "Architecture Mismatch: Why Reuse is so Hard", IEEE Software V12, #6, pp17-26, November 1995.

[5] Garlan and Shaw, "An Introduction to Software Architecture," in Advances in Software Engineering and Knowledge Engineering, vol. I, World Scientific Publishing Company,1993.

[6] Parnas, D., "Information distribution aspects of design methodology," in proceedings of IFIP conference, 1971, North Holland Publishing Co.

[7] Barkmeyer, E., SIMA Reference Architecture Part I: Activity Models, NIST Technical Report (in publication).

[8] Abowd, G., Bass, L., Kazman, R., Webb, M., "SAAM: A Method for Analyzing the Properties of Software Architecture," in Proceedings of the 16th International Conference on Software Engineering, Italy, May 1994.

[9] Thomas, I., Nejmeh, B., "Definitions of tool integration f.or environments," IEEE Software 9(3), pp. 29-35, March 1992.

[10] Wasserman, A., "Tool integration in software engineering environments," in F. Long, ed., *Software Engineering Environments, Lecture Notes in Computer Science 467*, pp. 138-150, Springer-Verlag, Berlin, Germany, 1990.

[11] *Principles of CASE Tool Integration*, Alan Brown, et. al., Oxford University Press, 1994, ISBN 0-19-509478-6.

[12] Brown, A., Judd, R., Riddick, F., "Architectural issues in the design and implementation of an integrated toolkit for manufacturing engineering" in the International Journal of Computer Integrated Manufacturing.

[13] Wakeman, L. and Jowett, J., "PCTE: The Standards for Open Repositories", Prentice-Hall, 1993.

[14] *Structural Modeling: An Application Framework and Development Process for Flight Simulators*, Gregory Abowd, Bass, L., Howard, L., Northrop, L., SEI Technical Report, CMU/SEI-93-TR-14, 1993, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

[15] A Software Architecture for Dependable and Evolvable Industrial Computing Systems, Sha, L., Rajkumar, R., Gagliardi, M., SEI Technical Report, CMU/SEI-95-TR-005, 1995, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.